On Callgraphs and Generative Mechanisms

Daniel Bilar

Wellesley College
Department of Computer Science
Wellesley, MA 02481, USA
dbilar@wellesley.edu

Abstract. This paper examines the structural features of callgraphs. The sample consisted of 120 malicious and 280 non-malicious executables. Pareto models were fitted to in-degree, out-degree and basic block count distribution, and a statistically significant difference shown for the derived power law exponent. A two-step optimization process involving human designers and code compilers is proposed to account for these structural features of executables.

1 Introduction

All commercial antivirus (AV) products rely on signature matching; the bulk of which constitutes strict byte sequence pattern matching. For modern, evolving *polymorphic* and *metamorphic* malware, this approach is unsatifactory. Clementi recently checked fifteen state-of-the-art, updated AV scanner against ten highly polymorphic malware samples and found false negative rates from 0-90%, with an average of 48% [9]. This development was already predicted in 2001 [51].

Polymorphic malware contain decryption routines which decrypt encrypted constant parts of the malware body. The malware can mutate its decryptors in subsequent generations, thereby complicating signature-based detection approaches. The decrypted body, however, remains constant. Metamorphic malware generally do not use encryption, but are able to mutate their body in subsequent generation using various techniques, such as junk insertion, semantic NOPs, code transposition, equivalent instruction substitution and register reassignments [8][49]. For a recent formalization of these code mutation techniques, the technical reader is referred to [17]. The net result of these techniques is a shrinking usable "constant base" for strict signature-based detection approaches.

Since signature-based approaches are quite fast (but show little tolerance for metamorphic and polymorphic code) and heuristics such as emulation are more resilient (but quite slow and may hinge on environmental triggers), a detection approach that combines the best of both worlds would be desirable. This is the philosophy behind a structural fingerprint. Structural fingerprints are statistical in nature, and as such are positioned as 'fuzzier' metrics between static signatures and dynamic heuristics. The structural fingerprint investigated in this paper for differentiation purposes is based on some properties of the executable's callgraph.

The rest of this paper is structured as follows: Section 2 describes the setup, data, procedures and results. Section 3 gives a short overview of related work on graph-based classification. Section 4 sketches the proposed generative mechanism.

2 Generating the callgraph

Primary tools used are described in more details in the Acknowledgements.

2.1 Samples

For non-malicious software, henceforth called 'goodware', sampling followed a two-step process: We inventoried all PEs (the primary 32-bit Windows file format) on a Microsoft XP Home SP2 laptop, extracted uniform randomly 300 samples, discarded overly large and small files, yielding 280 samples. For malicious software (malware), seven classes of interest were fixed: backdoor, hacking tools, DoS, trojans, exploits, virus, and worms. The worm class was further divided into Peer-to-Peer (P2P), Internet Relay Chat/Instant Messenger (IRC/IM), Email and Network worm subclasses. For an non-specialist introduction to malicious software, see [48]; for a canonical reference, see [50].

Each class (subclass) contained at least 15 samples. Since AV vendors were hesitant for liability reasons to provide samples, we gathered them from herm1t's collection [24] and identified compiler and (potential) packer metadata using PEiD. Practically all malware samples were identified as having been compiled by MS C++ 5.0/6.0, MS Visual Basic 5.0/6.0 or LCC, and about a dozen samples were packed with various versions of UPX (an executable compression program). Malware was run through best-of-breed, updated open- and closed-source AV products yielding a false negative rate of 32% (open-source) and 2% (closed-source), respectively. Overall file sizes for both mal- and goodware ranged from Θ(10kb) to Θ(1MB)[1]. A preliminary file size distribution investigation yielded a log-normal distribution; for a putative explanation of the underlying generative process, see [38] and [31].

All 400 samples were loaded into the de-facto industry standard disassembler (IDA Pro [22]), inter- and intra-procedurally parsed and augmented with symbolic meta-information gleaned programmatically from the binary via FLIRT signatures (when applicable). We exported the identified structures exported via IDAPython into a MySQL database. These structures were subsequently parsed by a disassembly visualization tool (BinNavi [13]) to generate and investigate the callgraph.

[1] A function $f(n)$ is $\Theta(g(n))$ if there are positive constants c_1, c_2, and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$, $\forall n \geq n_0$. See [26] for a readable discussion of asymptotic notation etymology.

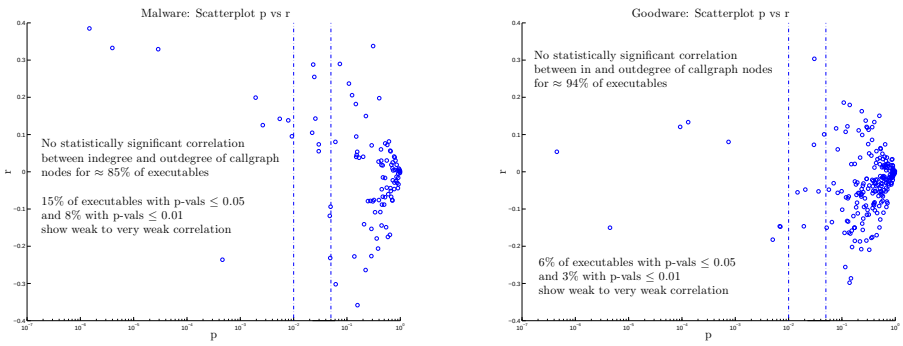
2.2 Callgraph

Following [14], we treat an executable as a *graph of graphs*. This follows the intuition that in any procedural language, the source code is structured into *functions* (which can be viewed as a flowchart, e.g. a directed graph which we call *flowgraph*). These functions call each other, thus creating a larger graph where each node is a function and the edges are calls-to relations between the functions. We call this larger graph the *callgraph*. We recover this structure by diassembling the executable into individual instructions. We distinguish between *short* and *far* branch instructions: Short branches do not save a return address while far branches do. Intuitively, short branches are normally used to pass control around within one function of the program, while far branches are used to call other functions.

A sequence of instructions that is continuous (e.g. has no branches jumping into its middle and ends at a branch instruction) is called a *basic block*. We consider the graph formed by having each basic block as a node, and each short branch an edge. The connected components in this directed graph correspond to the flowgraphs of the functions in the source code. For each connected component in the previous graph, we create a node in the callgraph. For each far branch in the connected component, we add an edge to the node corresponding to the connected component this branch is targeting. Figs. 7 and 8 in the Appendix illustrate a function's flow- and callgraph, respectively.

Formally, denote a callgraph CG as $CG = G(V, E)$, where $G(\cdot)$ stands for 'Graph'. Let $V = \bigcup F$, where $F \in normal, import, library, thunk$. This just says that each function in CG is either a 'library' function (from an external libraries statically linked in), an 'import' function (dynamically imported from a dynamic library), a 'thunk' function (mostly one-line wrapper functions used for calling convention or type conversion) or a 'normal' function (can be viewed as the executables own function). Following metrics were programmatically collected from CG

- $|V|$ is number of nodes in CG, i.e the function count of the callgraph
- For any $f \in V$, let $f = G(V_f, E_f)$ where $b \in V_f$ is a block of code, i.e each node in the callgraph is itself a graph, a flowgraph, and each node on the flowgraph is a basic block
- Define $IC : B \to N$ where B is defined to be set of blocks of code, and $IC(b)$ is the number of instructions in b. We denote this function shorthand as $|b|_{IC}$, the number of instructions in basic block b.
- We extend this notation $|\cdot|_{IC}$ to elements of V be defining $|f|_{IC} = \sum_{b \in V_f} |b|_{IC}$. This gives us the total number of instructions in a node of the callgraph, i.e in a function.
- Let $d^+_G(f)$, $d^-_G(f)$ and $d^{bb}_G(f)$ denote the indegree, outdegree and basic block count of a function, respectively.



(a) Malware: p vs $r_{in,out}$ (b) Goodware: p vs $r_{in,out}$

Fig. 1. Correlation coefficient $r_{in,out}$

2.3 Correlations

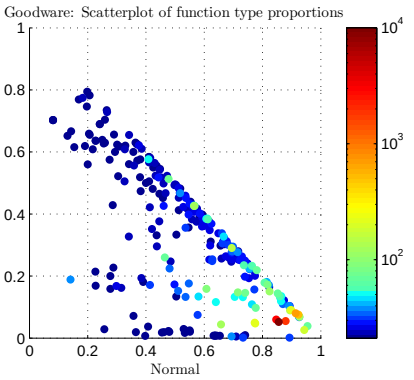
We calculated the correlation between in and outdegree of functions. Prior analysis of static class collaboration networks [44][40] suggest an anti-correlation, characterizing some functions as source or sinks. We found no significant correlation between in and outdegree of functions in the disassembled executables (Fig. 1). Correlation intuitively is unlikely to occur except in the '0 outdegree' case (the BinNavi toolset does not generate the flowgraph for imported functions, i.e. an imported function automatically has outdegree 0, and but will be called from many other functions).

Additionally, we size-blocked both sample groups into three function count blocks, with block criteria chosen as $\Theta(10)$, $\Theta(100)$ and $\Theta(1000)$ function counts to investigate a correlation between instruction count in functions and complexity of the executable (with function count as a proxy). Again, we found no correlation at significance level ≤ 0.001. Coefficient values and the IQR for instruction counts are given in Table 1. IQR, short for Inter-Quartile Range, is a spread measure denoting the difference between the 75^{th} and the 25^{th} percentiles of the sample values.

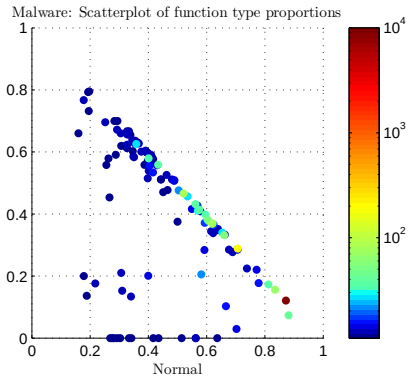
The first result corroborate previous findings; the second result at the phenomenological level agrees with the 'refactoring' model in [40], which posits that excessively long functions that tend to be decomposed into smaller functions. Remarkably, the spread is quite low, on the order of a few dozen instructions. We will discuss models more in section 4.

2.4 Function types

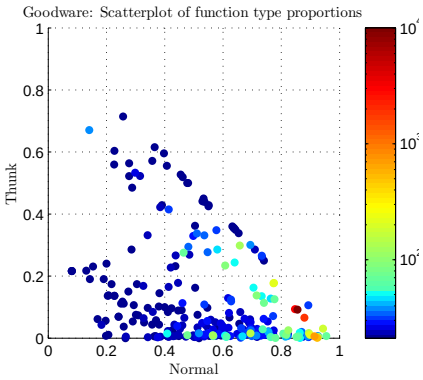
Each point in the scatterplots in Fig. 2 represents three metrics for one individual executable: Function count, and the proportions of normal function, static library + dynamic import functions, and thunks. Proportions for an individual executable add up to 1. The four subgraphs are parsed thusly, using Fig. 2(c)



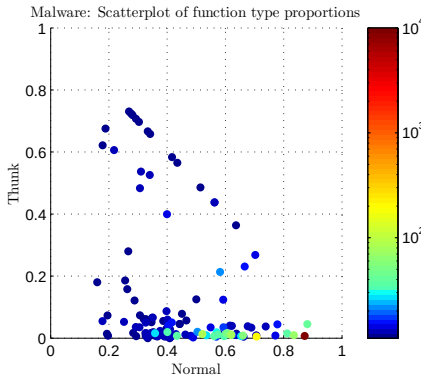
(a) GW:Norm vs Lib+Imp



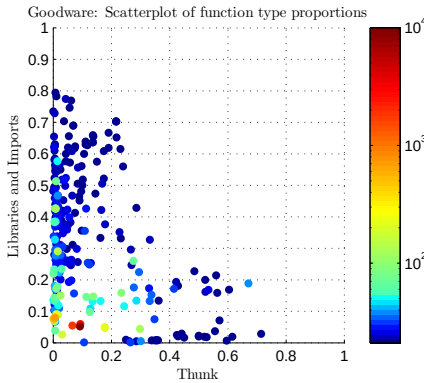
(b) MW:Norm vs Lib+Imp



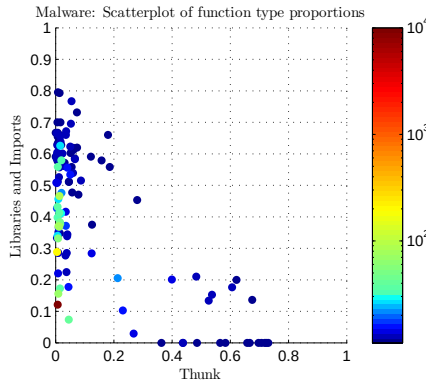
(c) GW:Norm vs Thunk



(d) MW:Norm vs Thunk



(e) GW:Thunk vs Lib+Imp



(f) MW:Thunk vs Lib+Imp

Fig. 2. Scatterplot of function type proportions

class	metric	$\Theta(10)$	$\Theta(100)$	$\Theta(1000)$
Goodware	r	0.05	-0.017	-0.0366
	IQR	12	44	36
Malware	r	0.08	0.0025	0.0317
	IQR	8	45	28

Table 1. Correlation, IQR for instruction count

as an example. The x-axis denotes the proportion of 'normal' function, and the y-axis the proportion of "thunk" functions in the binaries. The color of each point indicates $|V|$, which may serve as a rough proxy for the executable's size. The dark red point at (X, Y)= `(0.87, 0.007)` is `endnote.exe`, since it is the only goodware binary with functions count of $\Theta(10^4)$.

Most thunks are wrappers around imports, hence in small executables, a larger proportion of the functions will be thunks. The same holds for libraries: The larger the executable, the smaller the percentage of libraries. This is heavily influenced by the choice of dynamic vs. static linking. The thunk/library plot, listed for completeness reasons, does not give much information, confirming the intuition that they are independent of each other, mostly due to compiler behavior.

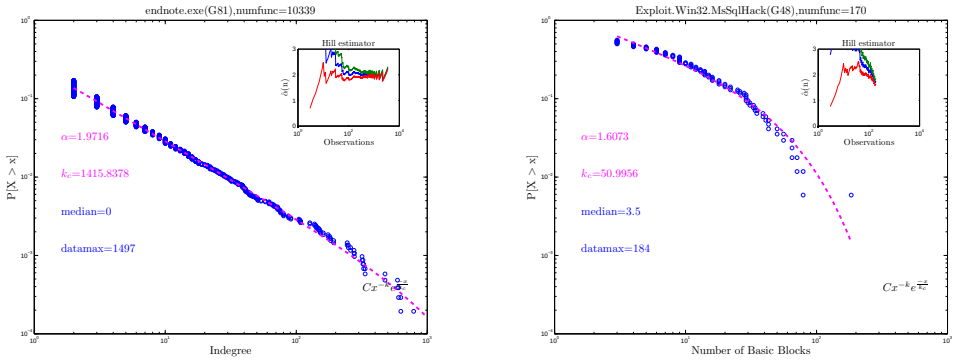
2.5 α fitting with Hill estimator

Taking my cue from [43] who surveyed empirical studies of technological, social, and biological networks, we hypothesize that the discrete distributions of $d^+(f), d^-(f)$ and $d^{bb}(f)$ follows a truncated power law of the form $P_{d^\star(f)}(m) \sim m^{\alpha_{d^\star(f)}} e^{-\frac{m}{k_c}}$, where k_c indicates the end of the power law regime. Shorthand, we call $\alpha_{d^\star(f)}$ for the respective metrics α_{indeg}, α_{outdeg} and α_{bb}.

Figs. 3(a) and 3(b) show *pars pro toto* the fitting procedures for our 400 samples. The plot is an Empirical Complimentary Cumulative Density Function Plot (ECCDF). A cumulative distribution function (CDF) $F(x) = P[X \leq x]$ of a random variable X denotes the probability that the observed value of X is at most x. 'Complimentary' simply represents the CDF as $1 - F(x)$, whereas the prefix 'empirical' signifies that experimental samples generated this (step) function.

The x-axis show indegree,the y-axis show the ECCDF P[X>x] that a function in `endote.exe` has indegree of x. If P[X>x] can be shown to fit a Pareto distribution, we can extract the power law exponent for PMF $P_{d^\star(f)}(m)$ from the CDF fit (see [1] and more extensively [41] for the relationship between Pareto, power laws and Zipf distributions). The probability mass function PMF $p[X = x]$ denotes the probability that a discrete random variable X takes on value x.

Parsing Fig. 3(a)): Blue points denotes the data points (functions) and two descriptive statistics (median and the maximum value) for the indegree distribution for `endote.exe`. We see that for `endnote.exe`, 80% of functions have



(a) GW sample: Fitting α_{indeg} and k_c

(b) MW sample: Fitting α_{bb} and k_c

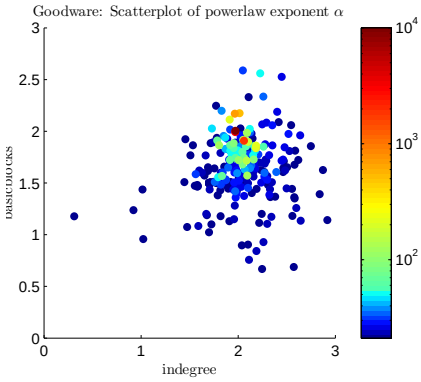
Fig. 3. Pareto fitting ECCDFs, shown with Hill estimator inset

a indegree=1, 2% indegree >10. and roughly 1% indegree > 20. The fitted distribution is shown in magenta, together with the parameters $\alpha = 1.97$ and $k_c = 1415.83$.

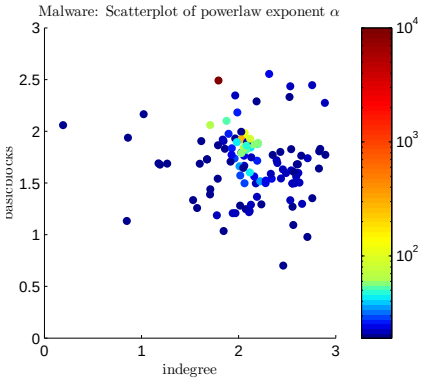
Although tempting, simply 'eyeballing' Pareto CDFs for the requisite linearity on a log-log scale [21] is not enough: Following [38] on philosophy and [46] on methodology, we calculate the Hill estimator$\hat{\alpha}$ whose asymptotical normality is then used to compute a 95% CI. This is shown in the inset and serves as a Pareto model self-consistency check that estimates the parameter α as a function of the number of observations. As the number of observations i increase, a model that is consistent along the data should show roughly $CI_i \supseteq CI_{i+1}$. For an insightful exposé and more recent procedures to estimate Pareto tails, see [56][16].

To tentatively corroborate the consistency of our posited Pareto model, 30 (goodware) and 21 (malware) indegree, outdegree and basic block ECCDF plots were uniformly sampled into three function count blocks, with block criteria chosen as $\Theta(10)$, $\Theta(100)$ and $\Theta(1000)$ function counts, yielding a sampling coverage of 10 %(goodware) and 17%(malware). Visual inspection indicates that for malware, the model seemed more consistent for outdegree than indegree at all function sizes. For basic block count, the consistency tends to be better for smaller executables. We see these tendency for goodware, as well, with the observation that outdegree was most consistent in size block $\Theta(100)$; for $\Theta(10)$ and $\Theta(1000)$. For both malware and goodware, indegree seemed the least consistent, quite a few samples did exhibit a so-called 'Hill Horror Plot' [46], where $\hat{\alpha}$s and the corresponding CIs were very jittery.

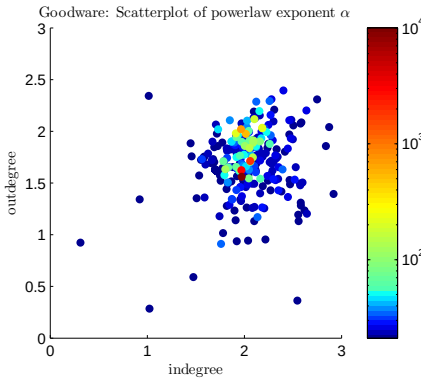
The fitted power-law exponents α_{indeg}, α_{outdeg}, α_{bb}, together with individual functions' callgraph size are shown Fig. 4. For both classes, the range extends for $\alpha_{indeg} \approx$ [1.5-3], $\alpha_{outdeg} \approx$ [1.1-2.5] and $\alpha_{bb} \approx$ [1.1-2.1], with a slightly greater spread for malware.



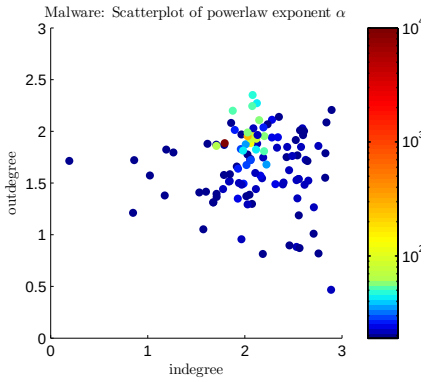
(a) GW:α_{indeg} vs α_{outdeg}



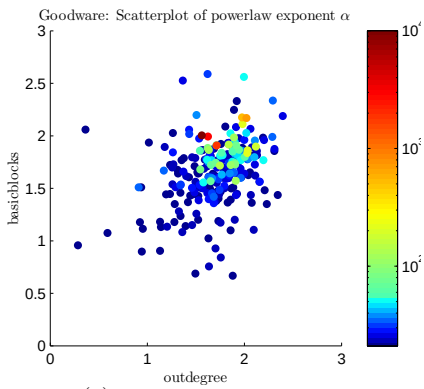
(b) MW:α_{indeg} vs α_{outdeg}



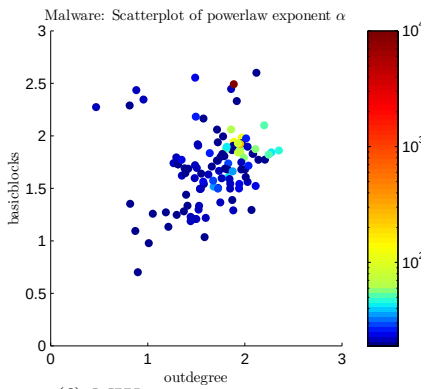
(c) GW:α_{indeg} vs α_{bb}



(d) MW:α_{indeg} vs α_{bb}



(e) α_{outdeg} vs α_{bbdeg}



(f) MW:α_{outdeg} vs α_{bbdeg}

Fig. 4. Scatterplots of α's

2.6 Testing for difference

We now check whether there are any statistically significant differences between (α, k_c) fit for goodware and malware, respectively. Following procedures in [57], We find α_{indeg}, α_{outdeg} and α_{bb} distributed approximately normal. The exponential cutoff parameters k_c are lognormally distributed. Applying a standard two-tailed t-test (Table 2), we find at significance level 0.05 ($t_{critical}$=1.97) only $\mu(\alpha_{bb,malware}) \geq \mu(\alpha_{bb,goodware})$.

For the basic blocks, $k_c \approx LogN(59.1, 52)$ (goodware) and $\approx LogN(54.2, 44)$ (malware) and $\mu(k_c(bb, malware)) = \mu(k_c(bb, goodware))$ was rejected via Wilcoxon Rank Sum with $z = 13.4$. The steeper slope of malware's α_{bb} imply that functions in malware tend to have a lower basic block count. This can be accounted for by the fact that malware tends to be simpler than most applications and operates without much interaction, hence fewer branches, hence fewer basic blocks. Malware tends to have limited functionality, and operate independently of input from user and the operating environment. Also, malware is usually not compiled with aggressive compiler optimization settings. Such a regime leads to more inlining and thus increases the basic block count of the individual functions. It may be possible, too, that malware authors tend to break functions into simpler components than 'regular' programmers. The smaller cutoff point for malware seems to corroborate this, as well, in that the power law relationship holds over a shorter range. However, this explanation should be regarded as speculative pending further investigation.

class	Basic Block	Indegree	Outdegree
GW	N(1.634,0.3)	N(2.02, 0.3)	N(1.69,0.307)
MW	N(1.7,0.3)	N(2.08,0.45)	N (1.68,0.35)
t	2.57	1.04	-0.47

Table 2. α distribution fitting and testing

3 Related work

A simple but effective graph-based signature set to characterize statically disassembled binaries was proposed by Flake [18]. For the purposes of similarity analysis, he assigned to each function a 3-tuple consisting of basic blocks count, count of branches, and count of calls. These sets were used to compare malware variants and localize changes; an in-depth discussion of the involved procedures can be found in [14]. For the purposes of worm detection, Kruegel [27] extracts control flow graphs from executable code in network streams, augments them with a colouring scheme, identifies k-connected subgraphs that are subsequently used as structural fingerprints.

Power-law relationships were reported in [52] [40] [53] [7]. Valverde et al [52] measured undirected graph properties of static class relationships for Java Development Framework 1.2 and a racing computer game, ProRally 2002. They found the $\alpha_{JDK} \approx 2.5-2.65$ for the two largest (N_1=1376, N_2=1364) connected components and $\alpha_{game} \approx 2.85 \pm 1.1$ for the game (N=1989). In the context of studying time series evolution of C/C++ compile-time "#include" dependency graphs, $\alpha_{in} \approx 0.97-1.22$ and an exponential outdegree distribution are reported. This asymmetry is not explained.

Focusing on the properties of directed graphs, Potanin et al [44] examined the binary heap during execution and took a snapshot of 60 graphs from 35 programs written in Java, Self, C++ and Lisp. They concluded that the distributions of incoming and outgoing object references followed a power law with $\alpha_{in} \approx 2.5$ and $\alpha_{out} \approx 3$. Myers [40] embarked on an extensive and careful analysis of six large collaboration networks (three C++ static class diagrams and three C callgraphs) and collected data on in/outdegree distribution, degree correlation, clustering and complexity evolution of individual classes through time. He found roughly similar results for the callgraphs, $\alpha_{in} \approx \alpha_{in} \approx 2.5$, and noted that it was more likely to find a function with many incoming links than outgoing ones.

More recently, Chatzigeorgiou et al [7] applied algebraic methods to identify, among other structures, heavily loaded 'manager' classes with high in- and outdegree in three static OO class graphs. In the spirit of classification through motifs in [35] and graphlets in [45], Chatzigeorgiou proposes a similarity-measure algorithm to detect Design Patterns [20], best-practices high level design structure whose presence manifest themselves in the form of tell-tale subgraphs.

Analysis of non-graph-based structural features of executables were undertaken by [30] [4] [54]. Li et al [30] used statistical 1-gram analysis of binary byte values to generate a fingerprint (a 'fileprint') for file type identification and classification purposes. At the semantically richer opcode level, Bilar [4] investigated and statistically compared opcode frequency distributions of malicious and non-malicious executables. Weber et al [54] start from the assumption that compiled binaries exhibit homogeneities with respect to several structural features such as instruction frequencies, instruction patterns, memory access, jumpcall distances, entropy metrics and byte-type probabilities and that tampering by malware would disturb these statistical homogeneities.

4 Optimization processes

In 2003, Myers [40], within the context of code evolvability, investigated how certain software engineering practices might alter graph topologies. He proposed a 'refactoring' model which was phenomenologically able to reproduced key features of source code callgraphs, among them the in and outdegree distributions. He noted that refactoring techniques could be rephrased as optimizations. Earlier and more explicitly, Valverde et al [52] speculated that multidimensional optimization processes might be the causative mechanism for graph topological features they unearthed. It has also been suggested in other venues that opti-

mization processes are the norm, even the driving force, for various physical, biological, ecological and engineered systems [15][47]. We share this particular outlook.

We hypothesize that the call-graph features described in the preceding sections may be the phenomenological signature of two distinct, domain-specific HOT (Highly Optimized Tolerance) optimization processes; one involving human designers and the other, code compilers. HOT mechanisms are processes that induce highly structured, complex systems (like a binary executable) through heuristics that seek to optimally allocate resources to limit event losses in an probabilistic environment [5].

4.1 Background

For a historical sketch of models and processes that induce graphs, the reader is referred to [42]; for a shorter, more up-to-date synopsis on power laws and distinctive generative mechanisms, including HOT, see [41]. Variations of the Yule process , pithily summarized as a 'rich-get-richer' scheme, are the most popular. Physicist Barabasi rediscovered and recoined the process as 'preferential attachment' [3], although the process discovery antedates him by at least forty years (its origins lay in explaining biological taxa). In some quarters of the physics community, power laws have also been taken as a signature of *emergent* complexity posited by critical phenomena such as phase transitions and chaotic bifurcation points [6].

The models derived from such a framework are mathematically compelling and very elegant in their generality; with little more than a couple of parameter adjustments, they are able at some phenomenological level to generate graphs whose aggregate statistics (sometimes provably, sometimes asymptotically) exhibit power-law distributions. Although these models offer a relatively simple, mathematically tractable approximation of some features of the system under study, we think that HOT models with their emphasis on *evolved and engineered* complexity through feedback, tradeoffs between objective functions and resource constraints situated in a probabilistic environment is a more natural and appropriate framework to represent the majority of real-life systems. We illustrate the pitfalls of a narrow focus on power law metrics without proper consideration of real-life domain specification, demands and constraints with Fig. 5 from [12]: Note that the degree sequence in subfig e) is the same for all subfig a)-d), yet the topological structure for subfigs. a)-d) is vastly different. Along these lines, domain experts have argued against 'emergent' complexity models in the cases of Internet router [29] and in river stream [25] structures.

4.2 Human design and coding as HOT mechanism

The first domain-specific mechanism that induces a cost-optimized, resource-constrained structure on the executable is the human element. Humans using various best-practice software development techniques [28][20] have to juggle at various stage of the design and coding stages: Evolvability vs specificity of the

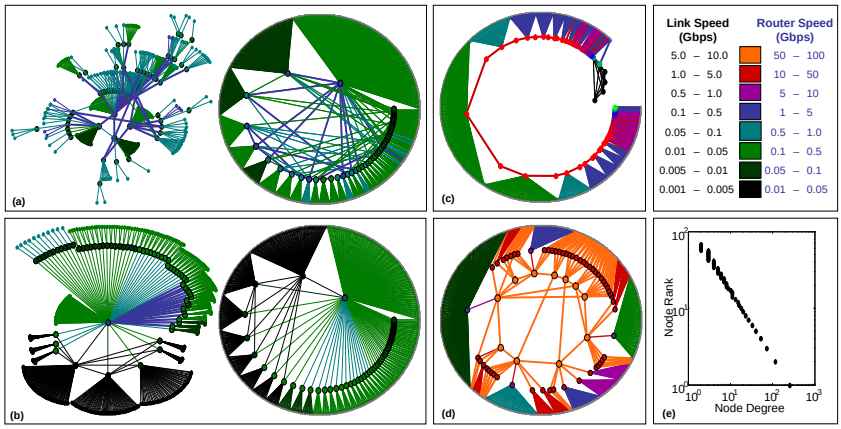
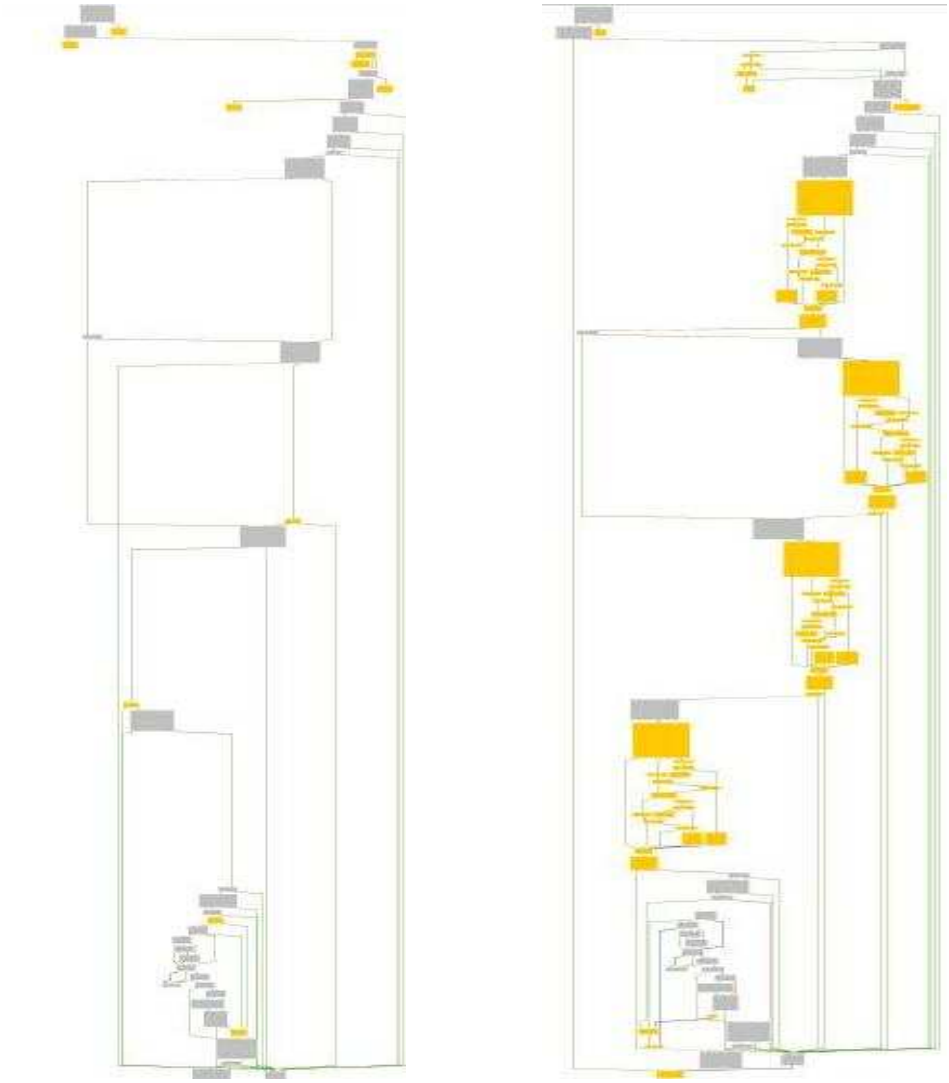


Fig. 5. Degree sequence e) following power law is identical for all graphs a)-d)

system, functionality vs code size, source readability vs development time, debugging time vs time-to-market, just to name a few conflicting objective function and resource constraints.

Humans design and implement programs against a set of constraints. For an involved discussion of software engineering practices and their relation to complex networks, the reader is referred to [40]. Designers take implicitly (rarely explicitly, though they should) the probability of the event space into consideration, indirectly through the choice of programming language (typed, OO, procedural, functional etc) and directly through the design choice of data structures and control flow. Human programmers generally design for average (or even optimal) operating environments; the resulting programs deal very badly with exceptional conditions effected by random inputs [34][33] and resource scarcity [55].

For years, the most common attack technique has been exploiting input validation vulnerabilities, accounting for over half of the published software vulnerabilities and over eighty percent of faults leading to successful penetration attacks. Miller et al testing Unix, Windows and OS X utilities [34][33] by subjecting them in the simplest case to random keyboard input, and reports crash failure rates of 25%-40%, 24%, and 7%, respectively. More recently, Whittaker et al [55] describe a dozen practical attack techniques targeting resources against which the executable were constrained (primarily by the human designer); among them memory, memory, disk space and network availability conditions. Effects of so-called "Reduction of Quality" attacks against optimizing control systems have also been studied by [37][36]. We shall give a toy example illustrating the 'attack-as-system perturbation-by-rare-events' view in Sec. 4.4.



(a) Compiler: CFG without loop unrolling (b) Compiler: CFG with loop unrolling

Fig. 6. Basic Block differences in CFG under compiler optimization regimes

4.3 Compiler as HOT mechanism

The second domain-specific mechanism that induces a cost-optimized, resource-constrained structure on the executable is the compiler. The compiler functions as a HOT process. Cost function here include memory footprint, execution cycles, and power consumption minimization, whereas the constraints typically involves register and cache line allocation, opcode sequence selection, number/stages of pipelines, ALU and FPU utilization. The interactions between at least 40+ optimization mechanisms (in itself a network graph [39, pp.326+]) are so complex that meta-optimizations [23] have been developed to heuristically choose a subset from the bewildering possibilities. Although the callgraph is largely invariant under most optimization regimes, more aggressive mechanisms can have a marked effect on callgraph structure. Fig. 6(a) shows a binary's CFG induced by the Intel C++ Compiler 9.1 under a standard optimization regime. The yellowed sections are loop structures. Fig. 6(b) shows the binary CFG of the same source code, but compiled under a more aggressive inlining regime. We see that the compiler unrolled the loops into an assortment of switch statements, vastly increasing the number of basic blocks, and hence changing the executable's structural features.

4.4 Example: PLR optimization problem as a HOT process

The HOT mechanism inducing the structure of the callgraph executable can be formulated as a Probability, Loss, Resource (PLR) optimization problem, which in its simplest form can be viewed as a generalization of Shannon source coding for data compression [32]. The reader is referred to [11] for details; I just give a sketch of the general formulation and a motivating example:

$$\min J \tag{1}$$

subject to

$$\sum r_i \leq R \tag{2}$$

where

$$J = \sum p_i l_i \tag{3}$$

$$l_i = f(r_i) \tag{4}$$

$$1 \leq i \leq N \tag{5}$$

We have a set of N events (Eq. 5) with occurring iid with probability p_i incurring loss l_i (Eq. 3), the sum-product of which is our objective function to be minimized (Eq. 1).Resources r_i are hedged against losses l_i (Eq. 4), subjected to resource bounds R (Eq. 2). We will demonstrate the applicability of this PLR model with the following short C program, adapted from [19]:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int provePequalsNP()
{
/* Next paper .. */
}
int bof()
{
char buffer[8]; /* an 8 byte character buffer */
strcpy(buffer,gets()); /*get input from the user*/
/* may not return if buffer overflowed
return 42;
}

int main(int argc, char **argv)
{
bof(); /*call bof() function*/
/* execution may never reach
next function because of overflow*/
provePequalsNP();
return 1000000;  /*exit with Clay prize*/
}
}
```

We assume here that the uncertain, probabilistic environment is just the user. She is asked for input in `gets()`, this represents the event. In the `C` code, the human designer specified an 8 byte buffer (`char buffer[8]`) and the compiler would dutifully allocate the minimum buffer needed for 8 bytes (this is the resource r). Hence, the constrained resources r is the variable `buffer`. The loss associated with the user input event is really a step function; as long as the user satisfies the assumption of the designer, the 'loss' is constant, and can be seen (simplified) as just the 'normal' loss incurred in proper continuation of control flow. Put differently, as long as user input is ≤ 8 bytes, the resource r is minimally sufficient to ensure normal control flow continuation. If, however, the user decides to input 'Superfragilisticexpialidocious' (which was implicitly assumed to be an unlikely/impossible event by the human designer in the code declaration), the loss l takes a huge jump: a catastrophic loss ensues since `strcpy(buffer,gets())` overflows `buffer`. The improbable event breaches the resource and now, control flow may be rerouted, the process crashed, shellcode executed via a stack overflow (or in our example, fame remains elusive). This is a classic buffer overflow attack and the essence of hacking in general - violating assumptions by 'breaking through' the associated resource allocated explicitly (input validation) and implicitly (race condition attacks, for instance) by the programmer, compiler or at runtime by the OS.

What could have prevented this catastrophic loss? A type-safe language such as Java and C# rather than C, more resources in terms of buffer space and more

code in terms of bounds checking from the human designer's side theoretically would have worked. In practice, for a variety of reasons, programmers write unsafe, buggy code. Recently, compiler guard techniques [10] have been developed to make these types of system perturbation attacks against allocated resources harder to execute or more easily to detect; again attacks against these compiler guard techniques have been developed [2].

5 Conclusion

We started by analyzing the callgraph structure of 120 malicious and 280 non-malicious executables, extracting descriptive graph metrics to assess whether statistically relevant differences could be found. Malware tends to have a lower basic block count, implying a simpler structure (less interaction, fewer branches, limited functionality). The metrics under investigation were fitted relatively successfully to a Pareto model. The power-laws evidenced in the binary call-graph structure may be the result of optimization processes which take objective function tradeoffs and resource constraints into account. In the case of the callgraph, the primary optimizer is the human designer, although under aggressive optimization regimes, the compiler will alter the callgraph, as well.

6 Appendix

We illustrate the concept of a flowgraph, callgraph and basic block by means of a fragment dissassembly[2] of Backdoor.Win32.Livup.c. We focus on the function `sub_402400`, consisting of six basic blocks. The flowgraph is given in Fig. 7. The assembly code for one basic block starting at `0x402486` and ending with a `jz` at `0x4024B9` is given below. Fig 8 shows the callgraph of `sub_402400`.

```
loc_402486:
402486  push    (0x4143E4, 4277220)
40248B  push    ebx
40248C  lea     eax, ss [esp+var_14]
402490  push    eax
402491  mov     ss [ebp+(0x14, 20)], edi
402494  mov     ss [ebp+(0x18, 24)], edi
402497  call    cs sub_402210
40249C  push    eax
40249D  lea     ecx, ss [ebp+(0x1c, 28)]
4024A0  mov     byte ss [esp+var_4], byte 2
4024A5  call    cs sub_401570
4024AA  mov     eax, ss [esp+var_14]
4024AE  mov     edx, ds [off_419064]
4024B4  lea     ecx, ds [eax + (0xFFFFFFF4, 4294967284)]
```

[2] See `www.viruslist.com/en/viruses/encyclopedia?virusid=44936` for more information.

```
4024B7   cmp      ecx, edx
4024B9   jz       byte cs loc_4024D9
```

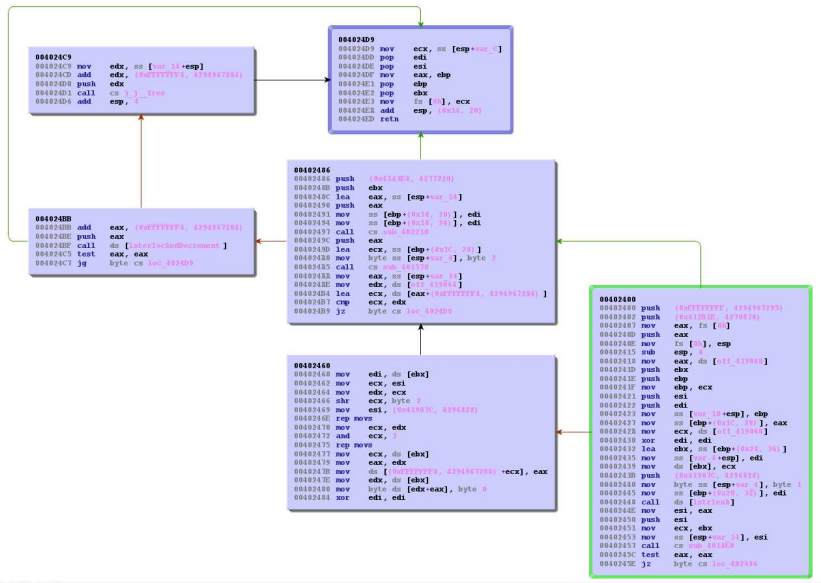


Fig. 7. Flowgraph for function `sub_402400`

Acknowledgements

This paper is an extended and revised version of a paper to appear in the European Journal on Artificial Intelligence's special issue on "Network Analysis in Natural Sciences and Engineering".

The goodware samples were indexed, collected and meta-data identified using *Index Your Files - Revolution! 3.1*, *Advanced Data Catalog 1.51* and *PEiD 0.94*, all freely available from `www.softpedia.com`. The executable's callgraph generation and structural identification was done with *IDA Pro 5* and a prerelease of *BinNavi 1.2*, both commercially available at `www.datarescue.com` and `www.sabre-security.com`. Programming was done with Python 2.4, freely available at `www.python.org`. Graphs generated with and some analytical tools provided by *Matlab 7.3*, commercially available at `www.matlab.com`.

We would like to thank Thomas Dullien (SABRE GmbH) without whose superb tools and contributions this paper could not have been written. He served as the initial inspiration for this line of research. Furthermore, we would like to thank Walter Willinger (AT&T Research), Ero Carrera (SABRE), Jason Geffner (Microsoft Research), Scott Anderson, Frankly Turbak, Mark Sheldon, Randy

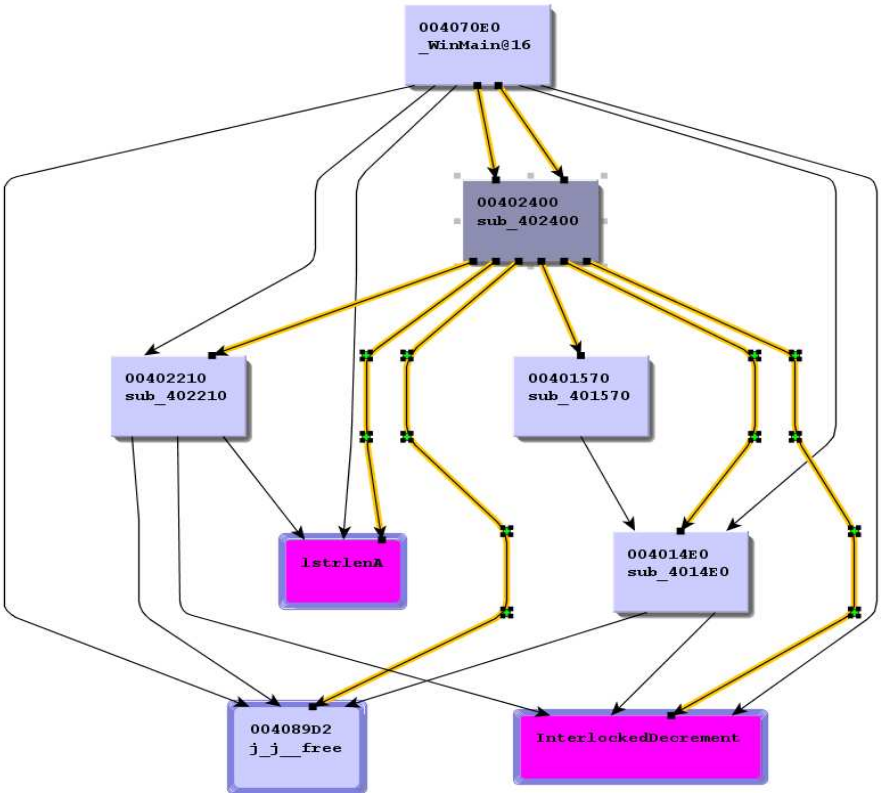


Fig. 8. Callgraph for function `sub_402400`

Shull, Frédéric Moisy (Université Paris-Sud 11), Mukhtar Ullah (Universität Rostock), David Alderson (Naval Post Graduate School), as well as the anonymous reviewers for their helpful comments, suggestions, explanations, and arguments.

References

1. L. Adamic and B. Huberman. Zipf's law and the internet. *Glottometrics*, 3:143–150, 2002.
2. S. Alexander. Defeating compiler-level buffer overflow protection. *j-LOGIN*, 30(3):59–71, June 2005.
3. A. L. Barabasi. Mean field theory for scale-free random networks. *Physica A Statistical Mechanics and its Applications*, 272:173–187, Oct 1999, cond-mat/9907068.
4. D. Bilar. Fingerprinting malicious code through statistical opcode analysis. In *ICGeS '07: Proceedings of the* 3^{rd} *International Conference on Global E-Security*, London (UK), April 2007.
5. J. M. Carlson and J. Doyle. Highly optimized tolerance: A mechanism for power laws in designed systems. *Physical Review E*, 60(2):1412+, 1999.

6. J. M. Carlson and J. Doyle. Complexity and robustness. *Proceedings of the National Academy of Sciences*, 99 Suppl 1:2538–2545, February 2002.
7. A. Chatzigeorgiou, N. Tsantalis, and G. Stephanides. Application of graph theory to OO software engineering. In *WISER '06: Proceedings of the 2006 international workshop on Workshop on interdisciplinary software engineering research*, pages 29–36, New York, NY, USA, 2006. ACM Press.
8. M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In *Security '03:Proceedings of the* 12^{th} *USENIX Security Symposium*, pages 169–186. USENIX Association, USENIX Association, August 2003.
9. A. Clementi. Anti-virus comparative no. 11. Technical report, Kompetenzzentrum IT, Insbruck (Austria), August 2006. `http://www.av-comparatives.org/seiten/ergebnisse/report11.pdf`.
10. C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, January 1998.
11. J. Doyle and J. M. Carlson. Power laws, highly optimized tolerance, and generalized source coding. *Physical Review Letters*, 84(24):5656–5659, June 2000.
12. J. C. Doyle, D. L. Alderson, L. Li, S. Low, M. Roughan, S. Shalunov, R. Tanaka, and W. Willinger. The "robust yet fragile" nature of the Internet. *Proceedings of the National Academy of Sciences*, 102(41):14497–14502, 2005.
13. T. Dullien. Binnavi v1.2. `http://www.sabre-security.com/products/binnavi.html`, 2006.
14. T. Dullien and R. Rolles. Graph-based comparison of executable objects. In *SSTIC '05: Symposium sur la Sécurité des Technologies de l'Information et des Communications*, Rennes, France, June 2005.
15. I. Ekeland. *The Best of All Possible Worlds: Mathematics and Destiny*. U Chicago Press, October 2006.
16. Z. Fan. *Estimation problems for distributions with heavy tails*. PhD thesis, Georg-August-Universität zu Göttingen, 2001.
17. É. Filiol. Metamorphism, formal grammars and undecidable code mutation. *International Journal of Computer Science*, 2(2):70–75, 2007.
18. H. Flake. Compare, Port, Navigate. Black Hat Europe 2005 Briefings and Training, March 2005.
19. J. C. Foster, V. Osipov, N. Bhalla, and N. Heinen. *Buffer Overflow Attacks*. Syngress, 2005.
20. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. *Lecture Notes in Computer Science*, 707:406–431, 1993.
21. M. L. Goldstein, S. A. Morris, and G. G. Yen. Problems with fitting to the power-law distribution. *European Journal of Physics B*, 41(2):255–258, September 2004, cond-mat/0402322.
22. I. Guilfanov. Ida pro v5.0.0.879. `http://www.datarescue.com/idabase/`, 2006.
23. M. Haneda, P. M. W. Knijnenburg, and H. A. G. Wijshoff. Optimizing general purpose compiler optimization. In *CF '05: Proceedings of the* 2^{nd} *conference on Computing Frontiers*, pages 180–188, New York, NY, USA, 2005. ACM Press.
24. herm1t. VX Heaven. `http://vx.netlux.org//`, 2007.
25. J. W. Kirchner. Statistical inevitability of horton's laws and the apparent randomness of stream channel networks. *Geology*, 21:591–594, 1993.
26. D. E. Knuth. Big omicron and big omega and big theta. *SIGACT News*, 8(2):18–24, 1976.

27. C. Krügel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In A. Valdes and D. Zamboni, editors, *Recent Advances in Intrusion Detection*, volume 3858 of *Lecture Notes in Computer Science*, pages 207–226. Springer, 2005.
28. J. Lakos. *Large-scale C++ software design.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1996.
29. L. Li, D. Alderson, W. Willinger, and J. Doyle. A first-principles approach to understanding the internet's router-level topology. In *SIGCOMM '04: Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 3–14, New York, NY, USA, 2004. ACM Press.
30. W.-J. Li, K. Wang, S. Stolfo, and B. Herzog. Fileprints: Identifying file types by n-gram analysis. In *SMC '05:Proceedings from the Sixth Annual IEEE Information Assurance Workshop on Systems, Man and Cybernetics*, pages 64– 71, West Point (NY), June 2005.
31. E. Limpert, W. A. Stahel, and M. Abbt. Log-normal distributions across the sciences: Keys and clues. *BioScience*, 51(5):341–352, May 2001.
32. M. Manning, J. M. Carlson, and J. Doyle. Highly optimized tolerance and power laws in dense and sparse resource regimes. *Physical Review E (Statistical, Nonlinear, and Soft Matter Physics)*, 72(1):016108–016125, July 2005, physics/0504136.
33. B. P. Miller, G. Cooksey, and F. Moore. An empirical study of the robustness of macos applications using random testing. In *RT '06: Proceedings of the 1st International workshop on Random testing*, pages 46–54, New York, NY, USA, 2006. ACM Press.
34. B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. *Communication of the ACM*, 33(12):32–44, 1990.
35. R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network Motifs: Simple Building Blocks of Complex Networks. *Science*, 298(5594):824–827, 2002.
36. A. B. Mina Guirguis and I. Matta. Reduction of quality (roq) attacks on dynamic load balancers: Vulnerability assessment and design tradeoffs. In *Infocom '07: Proceedings of the* 26^{th} *IEEE International Conference on Computer Communication*, Anchorage (AK), May 2007 (to appear).
37. I. M. Mina Guirguis, Azer Bestavros and Y. Zhang. Adversarial exploits of end-systems adaptation dynamics. *Journal of Parallel and Distributed Computing*, 2007 (to appear).
38. M. Mitzenmacher. Dynamic models for file sizes and double pareto distributions. *Internet Mathematics*, 1(3):305–334, 2004.
39. S. S. Muchnick. pages 326–327.
40. C. Myers. Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs. *Physical Review E (Statistical, Nonlinear, and Soft Matter Physics)*, 68(4):046116, 2003.
41. M. Newman. Power laws, Pareto distributions and Zipf's law. *Contemporary Physics*, 46(5):323–351, September 2005.
42. M. Newman, A.-L. Barabasi, and D. J. Watts. *The Structure and Dynamics of Networks: (Princeton Studies in Complexity).* Princeton University Press, April 2006.
43. M. E. J. Newman. The structure and function of complex networks. *SIAM Review*, 45:167, 2003.
44. A. Potanin, J. Noble, M. Frean, and R. Biddle. Scale-free geometry in oo programs. *Commununication of the ACM*, 48(5):99–103, 2005.

45. N. Pržulj. Biological network comparison using graphlet degree distribution. In *Proceedings of the 2006 European Conference on Computational Biology, ECCB '06*, Oxford, UK, 2006. Oxford University Press.
46. S. Resnick. Heavy tail modeling and teletraffic data. *Annals of Statistics*, 25(5):1805–1869, 1997.
47. E. D. Schneider and D. Sagan. *Into the Cool : Energy Flow, Thermodynamics, and Life*. University Of Chicago Press, June 2005.
48. E. Skoudis and L. Zeltser. *Malware: Fighting Malicious Code*. Prentice Hall PTR, Upper Saddle River (NJ), 2003.
49. P. Szor. *The Art of Computer Virus Research and Defense*, pages 252–293. In [50], February 2005.
50. P. Szor. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, Upper Saddle River (NJ), February 2005.
51. P. Szor and P. Ferrie. Hunting for metamorphic. In *VB '01: Proceedings of the* 11^{th} *Virus Bulletin Conference*, September 2001.
52. S. Valverde, R. Ferrer Cancho, and R. V. Solé. Scale-free networks from optimal design. *Europhysics Letters*, 60:512–517, November 2002, cond-mat/0204344.
53. S. Valverde and R. V. Sole. Logarithmic growth dynamics in software networks. *Europhysics Letters*, 72:5–12, November 2005, physics/0511064.
54. M. Weber, M. Schmid, M. Schatz, and D. Geyer. A toolkit for detecting and analyzing malicious software. In *ACSAC '02: Proceedings of the* 18^{th} *Annual Computer Security Applications Conference*, Washington (DC), 2002.
55. J. Whittaker and H. Thompson. *How to break Software security*. Addison Wesley (Pearson Education), June 2003.
56. W. Willinger, D. Alderson, J. C. Doyle, and L. Li. More normal than normal: scaling distributions and complex systems. In *WSC '04: Proceedings of the 36th conference on Winter simulation*, pages 130–141. Winter Simulation Conference, 2004.
57. G. T. Wu, S. L. Twomey, , and R. E. Thiers. Statistical evaluation of method-comparison data. *Clinical Chemistry*, 21(3):315–320, March 1975.